# Peephole Optimization of Asynchronous Macromodule Networks

Ganesh Gopalakrishnan, *Member, IEEE,* Prabhakar Kudva, *Associate Member, IEEE,* and Erik Brunvand

***Abstract*— Most high-level synthesis tools for asynchronous circuits take descriptions in concurrent hardware description languages and generate networks of *macromodules* or *handshake components*. In this paper, we propose a peephole optimizer for these networks. Our peephole optimizer first deduces an equivalent blackbox behavior for the network using Dill's trace-theoretic *parallel composition* operator. It then applies a new procedure called *burst-mode reduction* to obtain burst-mode machines from the deduced behavior. In a significant number of examples, our optimizer achieves gate-count improvements by a factor of five, and speed (cycle–time) improvements by a factor of two. Burst-mode reduction can be applied to any macromodule network that is delay insensitive as well as deterministic. A significant number of asynchronous circuits, especially those generated by asynchronous high-level synthesis tools, fall into this class, thus making our procedure widely applicable.**

***Index Terms*— Asynchronous circuits, burst-mode controllers, delay insensitivity, macromodules, peephole optimization, resynthesis.**

## I. Introduction

MANY digital circuits that we use are *reactive* and *control-intensive* in nature: they receive data values from the external world at unpredictable moments and have to efficiently perform a piece of computation for each data value received, where the computations and control decisions may take a data-dependent amount of time. If one uses the synchronous clocked design style for these circuits, one has to do considerable timing analysis to ensure that a significant amount of useful work is performed by the combinational circuit elements within all clock cycles in all control-flow situations and under all data patterns. In short, one must generate clock cycles only when needed, and must be able to fill the cycles generated with useful work. Though advanced clocking techniques in this area, such as distributed clocking methods [1], [2] and/or gated clocking [3] offer a solution to these problems, these techniques are not yet ready for widespread incorporation into general application-specific integrated circuit (ASIC) design. Another problem with using synchronous clocking for reactive circuits is that design changes can result in more work (compared to self-timed circuits) in regaining a feasible clocking schedule that is also optimal.

Asynchronous (self-timed) circuits are quite natural for realizing circuits of the reactive and control-intensive variety. Encouraging results are being obtained by many groups in designing self-timed circuits in this domain, e.g., in communications components used in multiprocessors [4], hardware to network portable electronic devices [5], and digital signal-processing algorithms used in audio-electronics hardware [6].

However, for the sake of balanced comparison, we must point out that much more work is needed in making sure that asynchronous circuits perform as well as synchronous circuits do, are as easily testable, and can be easily integrated into existing design flows. Many groups are in active pursuit of these goals. Last, but not least, we must also ensure that system optimization techniques of proven value in the synchronous circuit domain must also be available in the self-timed circuit domain. In the context of this point, our paper addresses a major shortcoming in the spectrum of asynchronous controller realization methods. More specifically, we offer a way to improve control circuits generated by asynchronous high-level synthesis tools through *peephole optimization*.

The importance of peephole optimization stems from several sources. In the last decade, asynchronous circuit and system design has seen significant growth in the number of practitioners and a corresponding broadening of the basic understanding at both the practical and theoretical levels [7]. The result is that there are numerous design styles, many of which are supported by reasonable synthesis and analysis tools. In order to facilitate the design of asynchronous circuits, several groups [8]–[10] have developed *high-level synthesis tools* that translate concurrent program-like descriptions (semi-) automatically into asynchronous circuits. These tools take hardware descriptions in a concurrent process description language and translate them into networks of *macromodules* [11]–[14].

Macromodules are hardware primitives that have an area complexity of anywhere from one to several tens of two-input gate equivalents, and are designed to support common control-flow constructs. Some examples of commonly used macromodules are the *procedure call* element (CALL), the *control-flow merge* element (MERGE), the *control-flow join* element (JOIN), the *sequencer* element (SEQUENCER), the *toggle* element (TOGGLE), various arbiters (ARBITER), modules that alter control-flow based on Boolean conditions (SELECT), and modules that help implement finite-state machines (FSM's) (such as DECISION-WAIT).

Manuscript received May 26, 1995; revised July 24, 1998. The work of G. Gopalakrishnan and E. Brunvand were supported in part by the National Science Foundation under Award MIP 9215878. An earlier version of this paper was published in the *IEEE Proc. International Conference Computer Design* (ICCD), Boston, MA, October 1994, pp. 442–446.

G. Gopalakrishnan and E. Brunvand are with the Computer Science Department, University of Utah 84112 USA (e-mail: ganesh@cs.utah.edu).

P. Kudva is with IBM T. J. Watson Research Center, Yorktown Heights, NY 10598 USA (e-mail: kudva@watson.ibm.com).

Publisher Item Identifier S 1063-8210(99)01549-8.

For several reasons, macromodules are popular targets for asynchronous circuit compilers. Since most macromodules are *delay insensitive* (DI) [15], their behavior is quite easily characterized, both individually as well as collectively. Since macromodules are higher level primitives than gates, it is much easier to write asynchronous compilers that target them, as opposed to directly targeting gates or transistors. However, macromodule networks generated by high-level synthesis tools often have subnetworks containing redundancies. These redundancies often stem from program "idioms" at the source code level. Such subnetworks can often be replaced (peephole optimized) by more efficient (in terms of area and/or time) macromodule networks that are, as far as their environment is concerned, behaviorally indistinguishable from the original network.

Peephole optimization has been studied by Brunvand [16] as well as van Berkel [17]. Their approach replaces macromodule networks by more efficient macromodule networks with the help of an *a priori* fixed collection of rewrite rules. In this paper, we take a different approach to the peephole optimization problem. We reduce the overall behavior of the macromodule subnetwork being optimized to a *burst-mode machine*. A burst-mode machine is an FSM in which every state transition is enabled by a nonempty set of input events (an "input burst") that can occur in any order; whenever a transition is enabled, the machine generates a set of output events (an "output burst") associated with the transition and moves on to its next state. In recent years, a number of highly efficient controller design methods based on *burst-mode machines* have been proposed by a number of authors, including Davis *et al.* [18], Nowick *et al.* [19], [20], and Yun *et al.* [21]. The efficiency, as well as wide applicability of burst-mode machine controllers, has been validated through a number of real-world designs that the above authors have built [18], [22]. However, in order to be able to employ burst-mode machines as optimized replacements for macromodule subnetworks, the following questions must be addressed.

1) How do we compute the overall behavior of the macromodule subnetwork being optimized?
2) Can we always generate a burst-mode machine corresponding to the overall behavior of a macromodule subnetwork? Are the behaviors of the macromodule subnetwork and that of the burst-mode machine replacement exactly the same? If not, why are the differences acceptable?
3) How area and time efficient are the burst-mode machine replacements over the macromodule subnetworks being replaced?

Answers to these questions will be provided in this paper.

### A. Related Work and Comparisons with Our Proposal

The peephole-optimization problem for macromodule networks has been addressed in the past by Brunvand and van Berkel. As opposed to being confined to a finite set of rewrite rules, our approach is one of resynthesis and is applicable to the entire class of asynchronous networks, which are deterministic and DI (as shown in Section IV).

In terms of the theoretical basis for justifying the peephole optimization procedure, van Berkel identifies a *refinement ordering* among processes. Brunvand's optimization rules have, similarly, been formally justified based on the *strong conformance* relation [23]. The theoretical basis of burst-mode reduction is presented in Section IV.

The approach of Martin [24] altogether avoids the peephole-optimization problem by synthesizing logic equations directly from a textual intermediate form called *production rules*. The generality of Martin's logic synthesis method is not well understood. We believe that asynchronous high-level synthesis approaches which generate macromodule networks as intermediate form possess several advantages, including intuitiveness of the generated circuits and ease of validation of the compiler. Therefore, we prefer the approach of generating macromodule networks and later optimizing them.

### B. Organization

The remainder of this paper is organized as follows. In Section II, we present basic definitions. In Section III, we provide an illustration of our method on a simple example. In Section IV, we provide details of our work. In Section V, we present our results, as well as concluding remarks.

## II. Basic Definitions

### A. Basics of Burst-Mode Machines

Burst-mode machines are a subclass of asynchronous FSM's [18]. A burst-mode machine is a Mealy-style FSM in which every transition is labeled with pairs $(I, O)$ (written in the usual "$I/O$" notation) where $I$ is a nonempty set of polarized (rising or falling) signal transitions called the *input burst*, and $O$ is the output burst. Contrary to the original definition [18], we require that $O$ be nonempty. This requirement is consistent with our assumption of delay insensitivity of macromodules. The environment can also be given a burst-mode specification by *mirroring* [25].

For input bursts $Is_1$ and $Is_2$, labeling any two FSM transitions leaving a state $s$, neither must be a subset of the other. This enforces determinacy. The collection of input bursts leaving a state need not be exhaustive; those input bursts that are not explicitly specified are assumed to be illegal. If a state of a burst-mode machine can be entered via two separate FSM transitions, then the output bursts associated with these FSM transitions must not contain signal transitions of opposite polarities.

### B. Assumptions About Macromodule Networks Being Optimized

The macromodule network being optimized by our optimizer must not contain *arbiters* or other nondeterministic components, as our optimizer generates burst-mode machines which are deterministic. The network should also be initially quiescent and should attain quiescence after processing every successive input burst, as will be detailed later. Finally, the network must be DI.

### C. Basics of Trace Theory

We employ Dill's trace theory [25] to model the behavior of individual macromodules as well as to obtain the composite behavior of networks of macromodules through *parallel composition* and *hiding*. For the purposes of this paper, the following property of these operators is important: *parallel composition and hiding preserve delay insensitivity*. This property is important because given a network of DI modules, we can be assured that Dill's parallel composition and hiding operators will produce a single inferred behavior, which will also be DI. A proof sketch is provided in the Appendix.

### D. Udding's Conditions for Delay Insensitivity

Udding [15] has provided four necessary and sufficient conditions on traces that characterize delay insensitivity. Among these conditions, the ones relevant for this paper are now briefly outlined. Condition (a) is: "*if a module accepts (generates) two inputs (i.e., input signal transitions) $a$ and $b$ in the order $ab$, it must also accept (generate) them in the order $ba$.*" If this were not so, the wires leading to the module (which can have arbitrary delays) can reorder the transitions and present them to the module in the wrong order.

Condition (b) is: "*for input symbol $a$ and output symbol $b$, and for arbitrary trace $t$, if the behaviors $ta$ and $tb$ are legal for the module, then the behaviors $tab$ as well as $tba$ must also be legal.*" This is explained as follows. After processing $t$, the module has the choice of generating a $b$ and awaiting an $a$ or vice-versa (and, likewise, the environment). Suppose the module chooses to generate the output $b$. The environment has no immediate way of knowing this (due to arbitrary wire delays). In fact, the environment may "think" that the module is waiting for an $a$ (which is also legal for the module to do after a $t$). Therefore, the environment can go ahead and generate an $a$ even before it receives a $b$ from the module. The module will, therefore, end up seeing the sequence $tba$, which better be legal for the module. Similarly (due to mirroring) the environment must be ready to process trace $tab$. Since the trace sets of the module and environment are the same (except for the directions of the symbols involved) both $tab$ and $tba$ must be legal for the module.

## III. Illustration of Our Approach

For our peephole-optimizer to be applicable to a network, the network's joint behavior with its environment must obey the following restrictions. After power-up, the network must be *quiescent*—i.e., it must not produce any output signal transition before first consuming an input signal transition. In a quiescent state, the network consumes an input burst and, in response, produces an output burst in a finite amount of time. When the last transition of the output burst has been produced, the network is assumed to have attained its next state, where it is ready for the whole process to repeat. This mode of interaction between the network and its environment is called the *burst-mode* behavior, which is a special case of fundamental-mode operation [26].

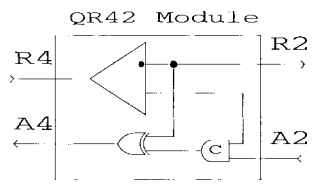


Fig. 1. A four-to-two quick-return converter.

The optimizer first obtains the overall behavior of the macromodule subnetwork being optimized using the composition operator on trace structures [25]. The behavior inferred in this fashion leaves out combinations of behaviors of the submodules that can never arise or can lead to internal hazards. The inferred behavior is converted into an encoded interface state graph (EISG) [27]. EISG's are automata that label their state transitions with polarized signal transitions. Finally, the optimizer converts the EISG into a burst-mode machine using "burst-mode reduction" (detailed later), and synthesizes the resulting burst-mode machine using an already available tool (e.g., see [18], [20], and [21]). Any delay-insensitive and deterministic module $M_{\mathrm{DI}}$ can be reduced (via burst-mode reduction) to a corresponding burst-mode machine $M_{\mathrm{BM}}$ in such a way that the operation of $M_{\mathrm{DI}}$ would be exactly the same as that of $M_{\mathrm{BM}}$, *provided* the environment obeys the fundamental-mode timing constraint associated with the burst-mode behavior.

Consider the example shown in Fig. 1. This subnetwork accepts a four-cycle handshake sequence [12] on `r4` and `a4` and generates a two-cycle handshake sequence [12] on `r2` and `a2`, with the property that some of the events in these handshake sequences can overlap in order to provide a high degree of concurrency, as shown by the Petri net in Fig. 2. Assume that all interface signals are low to begin with. When `r4+` occurs, the toggle element generates an `r2+`, as well as an `a4+` (through the exclusive OR (XOR) gate). Transition `a4+` is treated as the "ack" by the four-cycle side which generates an `a4` transition which, in turn, is forwarded by the toggle element to the upper input of the $C$-element as a rising signal transition. Meanwhile, `r2+` is treated as a request by the two-cycle side, which generates an `a2+`. This causes the $C$-element to receive rising transitions on both its inputs. Therefore, it generates an `a4−` through the XOR gate. A similar sequence of steps now ensues, during which the two-cycle interface returns to its initial state, and then the whole cycle repeats.

In optimizing QR42, we must first compose the behaviors of the three macromodules, shown in Fig. 1, along with the behaviors of "fictitious modules" that express the following constraints on its interfaces: the four-phase interface must witness a progression of events `r4; a4;` $\cdots$, and the two-phase interface must witnesses a progression of events `r2; a2;` $\cdots$. In addition, we must *hide* the internal signal that connects the lower output of the toggle element to the upper input of the $C$-element. The resulting blackbox behavior for QR42 is shown by the Petri net in Fig. 2. Next, we must obtain the state graph (EISG) corresponding to this Petri net. We can see that QR42 is initially quiescent, waiting for the singleton input-burst `r4`. After receiving `r4`, it has

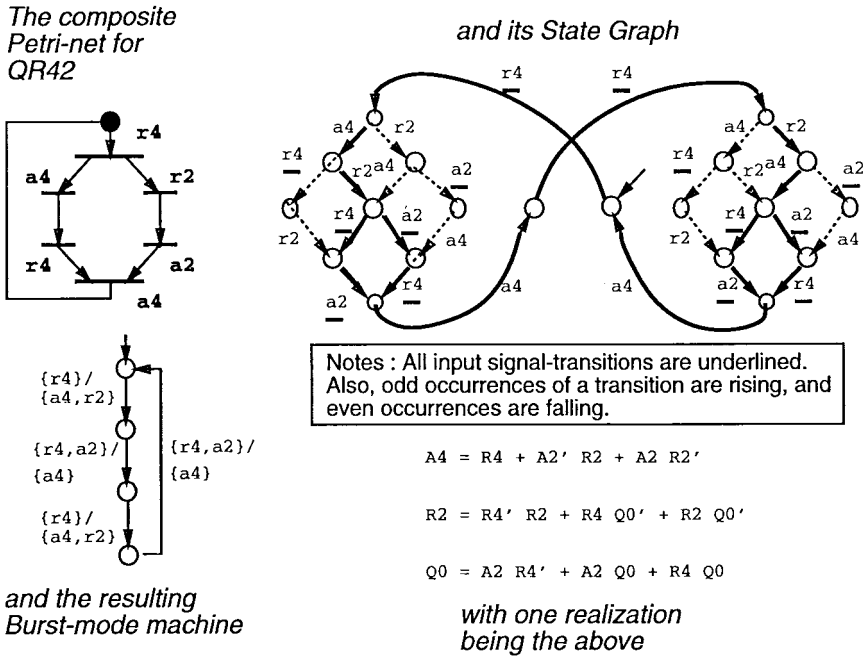


Fig. 2. Optimization of the four-to-two converter.

the option of producing a4 or r2. If a4 is produced first, QR42 is in a state where output r2 as well as input r4 are possible. If the environment were to follow the burst-mode behavior, however, it would first allow r2 to be produced before supplying the inputs r4 and a2 concurrently as an input burst. The important point to note is that even though a deterministic delay-insensitive module may possess a large number of behaviors, an environment that follows the burst-mode operating conditions invokes only a proper subset of these behaviors. In other words, the environment of each peephole applies the next set of inputs only after the peephole-optimized circuit has stabilized—a fact established through timing analysis.

We now perform *burst-mode reduction*, which retains only the heavy arrows in Fig. 2, and constructs the Burst-mode machine shown. We can ignore the dashed arrows because of the assumption of delay insensitivity (for reasons given later). Finally, we can synthesize the burst-mode machine (in our case, using Yun's tool [21]) to obtain logic equations shown in the figure.

## IV. Details of the Optimizer

### A. Identifying a Subnetwork to Optimize

The macromodule network to be optimized is identified based on performance considerations. In addition, it is preferable to peephole optimize clusters of tightly interacting macromodules first, to prevent the EISG's from becoming exponentially large.

### B. Environmental Constraints

Once a subnetwork is identified, the environment of the subnetwork must be suitably specified to avoid obtaining too general a result. For example, each subnetwork can interact with its environment through either an *active* or *passive* channel. An active channel involves the output of a *request* transition followed by the receipt of an *acknowledge* transition. A passive channel awaits a *request* transition and then generates an *acknowledge* transition. Channel connections to the environment must not be left "dangling," as this would cause impossible behaviors to be considered by the parallel composition process. For example, for an active channel, we must stipulate that *acknowledge* will come only after a *request*. If this is not specified, the parallel composition operator will allow for the possibility of an *acknowledge* even before a *request*. These constraints are expressed by introducing fictitious modules that possess the required input/output (I/O) traces and effectively "close off" the dangling channel connections properly. Also, in most asynchronous high-level synthesis tools, connections to datapath elements resemble active channels. Therefore, connections to datapath elements are modeled exactly as "dangling" active channels are modeled.

### C. Composed Trace Structures to EISG

After the network being optimized has been composed into a single trace structure, its description is converted into an EISG. In Dill's AVER system [25], trace structures are represented as transition-style automata. In other words, the polarities of signal transitions are not explicitly maintained. These automata can, therefore, be converted into EISG's by exhaustively "simulating" all their possible moves until all their reachable configurations are covered. Earlier, we had illustrated such a state graph for the QR42 module in Fig. 2. This process also can result in state explosion, especially if many nested branches are involved. Fortunately, this has not proven to be a problem in the examples considered thus far.

### D. Burst-Mode Reduction

Once EISG's are obtained, they are converted into equivalent burst-mode machines by means of *burst-mode reduction*. The basic intuition behind this algorithm was already illustrated on the QR42 system in Fig. 2. Basically, this algorithm traverses a path of the state graph starting from the starting state, collecting input transitions occurring along the way into the set *input-burst*, until it encounters a state that has only arcs labeled by output-signal transitions exiting it. The traversal is continued, now forming the set *output-burst*, until a state that has only arcs labeled by input-signal transitions exiting it. A burst-mode machine transition is now formed, and the algorithm continues processing the rest of the state graph. The basic intuition behind *burst-mode reduction* is that whenever "lattice shapes" representing concurrency are encountered in the state graph, such lattices are collapsed into input and output bursts. (However, there are some additional details, which are given below.)

Now we detail burst-mode reduction and provide a correctness argument for it. (Note: for clarity, we present a somewhat inefficient algorithm below. Our implementation is equivalent to the following, but more efficient.)

**Input** An EISG, which is a state graph with circles denoting states, and arcs between states labeled by a single polarized transition of an input signal or an output signal. Only those EISG's obtained by composing macromodules obeying restrictions stated earlier are considered.

**Output** A burst-mode machine.

**Method**

1) Mark all states as "not visited," and call the starting state *current*.
2) This step addresses the collapsing of "lattice shapes" in the state graph. Specifically, if *current* has not been visited, mark it as visited. If *current* has an exit through at least one output transition, retain any arbitrary output transition, while eliminating all others. Call the destination of the retained transition as *current*, and continue with Step 2. Else (all exits are through input transitions) retain all the transitions out of *current*, and consider all their destination states to be *current*, in turn, and continue with Step 2 for these states.
3) (We reach here after the initial "transition elimination" portion of the algorithm is over.) Remove unreachable portions of the state graph.
4) Set the starting state of the state graph as *current*.
5) Go to the *current* state. It will have exits only through input transitions. (This invariant is initially true due to the quiescence of the starting state, and is preserved by the way the following loop will work.)
   Take any path out of *current* and traverse it, collecting input transitions encountered along the way into a set *input burst*. (We will never encounter a state in the interim that has both an input exit, as well as an output exit.) Continue collecting input transitions, until we encounter a state with exactly one output exit. Call this state *intermediate*.
6) Continue traversing along an arbitrary path from state *intermediate* collecting output transitions into a set *output burst* until a state which has no exits through an output transition is encountered. Call this state *next*.
7) Construct a burst-mode machine transition from *current* going to *next* labeled by *input burst/output burst*.
8) Repeat the procedure from Step 5 for all paths emanating from *current*.
9) Repeat Step 4, now treating all the states marked *next* as *current*, and until all states have been visited.
10) Eliminate all duplicate transitions in the burst-mode machine.

These steps can be easily seen in Fig. 2, where we first eliminate all but the transitions shown in heavy lines, and then form burst-mode transitions out of the retained transitions. The steps in the algorithm can be justified as follows.

1) In Step 2, the algorithm chooses an arbitrary output transition among competing output transitions. This is justified because by Udding's Condition (a), (as evidenced by the "lattice shape" of the state graph), the ignored outputs are guaranteed to appear later in sequence.
2) In Step 2, the algorithm "prefers" output transitions over input transitions. This is justified on the following two counts.
   a) Because of Udding's Condition (b), competing inputs and outputs are also guaranteed to appear in all possible orders. Therefore, even if an input transition is ignored when it competes with an output transition, that input transition will be offered later in sequence.
   b) Due to the burst-mode assumption, the environment must allow the output transitions to happen before it applies inputs to the system. This is why output transitions are "preferred over" input transitions.
3) After the state graph has been pruned on the basis of the above statements, we enter the phase of forming input and output bursts for burst-mode transitions. In this process, it is not necessary to consider the particular order in which inputs or outputs appear in sequence. This is because a delay-insensitive system cannot count on inputs/outputs appearing in any particular order [Udding's Condition (a)].
4) In traversing from state *current* to state *intermediate*, if different sequences of input transitions coalesce into the same input burst set (e.g., `r4` and `a2` of QR42), then all these input transition sequences must lead to the same behavior from state *intermediate* onwards (in other words, all these input transition sequences must

cause the same set of output bursts and enter equivalent next states). This will result in duplicate burst-mode transitions that can be eliminated, as in Step 10.

### E. Correctness of Burst-Mode Reduction

The above reasoning shows that Burst-mode reduction results in a burst-mode machine that has the same behavior as the original macromodule network when that network is operated under the burst-mode assumption. The following well-formedness conditions of burst-mode machines are also guaranteed:

1) *Nonempty Input Bursts:* The fact that all input bursts are nonempty is guaranteed by the *quiescence* requirement that is an invariant of the loop beginning at state *current* in Step 4.
2) *Subset Property:* The subset property requires that no input burst can be a subset of another. This is true for the following reasons.
   a) In traversing from state *current* to state *intermediate* in Step 4, a sequence $s$ of inputs is collected to form the set *input burst*. Due to Udding's Condition (a), these inputs will appear in all permutations. Therefore, there can be no proper subsequence of $s$ that also goes between states *current* and *intermediate*.
   b) The other possibility is that a proper subsequence $s'$ of $s$ leads to a *different* state *intermediate*. Then, the state graph exhibits nondeterministic choice and, by definition, we cannot handle nondeterministic machines.
   c) The final possibility is that $s$ and $s'$ are identical and lead to the same state *intermediate*. This will result in duplicate transitions that get eliminated in Step 10.
3) *Unique Entry:* This is guaranteed by the way an EISG is generated. Essentially, a *state* of an EISG includes the state of the interface signals; hence, there cannot be a state conflict in the burst-mode machine because the EISG will allocate two separate states for noncompatible interface-signal assignments.

## V. Results and Concluding Remarks

A prototype implementation of the optimizer described here has been applied to a number of circuits (Fig. 3). The organization of the optimizer is shown in Fig. 4. The description of the macromodule circuit to be peephole optimized is accepted in the input format of the AVER system [28]. The overall behavior of this macromodule circuit is inferred from its structural description and the behaviors of its submodules using parallel composition. The output of this phase is a composite simple prefix-closed trace structure (SPCTS) [28]. This structure is then expanded into an EISG, which is then converted into a burst-mode machine via burst-mode reduction. The output of this phase is a burst-mode machine specification. Using Yun's three-dimensional (3-D) synthesis tools [21], we then obtain logic equations for the final implementation. Yun's 3-D tools also generate a Verilog [29] description for use by existing logic- and gate-level synthesis tools.

| Circuits | BM size (gates) | MM size (gates) | BM speed (gates) | MM speed (gates) |
|---|---|---|---|---|
| QR42 (hand) | 13 | 13 | 10 | 15 |
| QR42 (v1) | 13 | 74 | 10 | 20 |
| QR42 (v2) | 13 | 60 | 10 | 15 |
| Call2 | 17 | 21 | 4 | 8 |
| Call-C Idiom | 27 | 27 | 10 | 11 |
| Decision Wait (2x1) | 26 | 18 | 8 | 15 |
| Simple GVT (p.1) | 15 | 18 | 4 | 4 |
| Simple GVT (p.2) | 8 | 12 | 10 | 14 |
| Call3-Merge Optimization | 68 | 45 | 11 | 16 |
| Completion-tree size 3 | 8 | 10 | 6 | 10 |
| Completion-tree size 4 | 11 | 15 | 8 | 13 |
| Completion-tree size 5 | 14 | 20 | 8 | 17 |

Fig. 3. Performance of our optimizer.

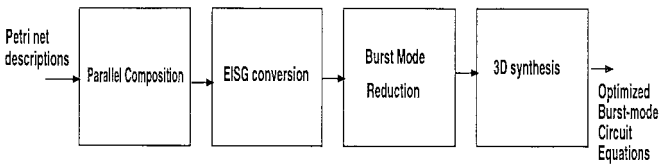


Fig. 4. Peephole optimization system.

For our procedure to perform efficiently, as well as retain only those behaviors that will arise in the intended use of the circuit, it is necessary to obey the following precautions.

1) As discussed in Section III, environmental constraints must be modeled; otherwise, the parallel composition algorithm will take longer and the inferred behavior will have redundancies. For example, for the QR42 module, if we do not express the fact that an `r4` signal will be generated before an `a4` is generated, the behavior inferred for the QR42 circuit will also allow for an `a4` to be input even before an `r4` is generated—a behavior of no interest to the users of QR42.
2) The order in which parallel composition is carried out is crucial in terms of deciding how efficiently parallel composition works. The rule-of-thumb to follow is "compose hierarchically, most constraining behaviors first," where "constraining" is defined as any behavior

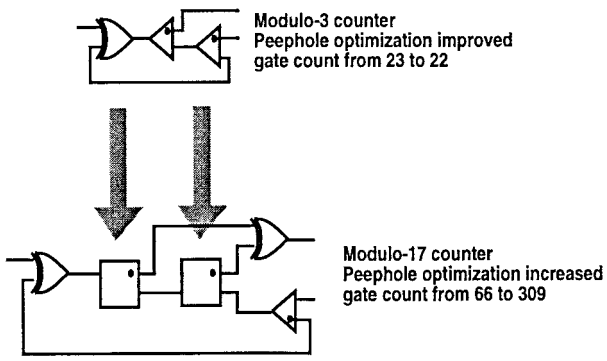


Fig. 5. An example of choosing different peephole sizes.

that forces things to occur in a certain way. For example, composing the behaviors of the two- and four-phase interfaces earlier in the parallel composition hierarchy can dramatically cut down the number of behaviors generated.

Though the execution times of the parallel composition tool and the burst-mode reduction program are worst-case exponential, our examples were processed quickly. In a significant number of examples, our optimizer achieves gate-count improvements by a factor of five, and speed (cycle-time) improvements by a factor of two as shown in Fig. 3. These examples were chosen based on the fact that they typify macromodule sub-circuits generated by asynchronous circuit compilers such as Occam [9] and SHILPA [10]. In some cases (e.g., decision wait), the gate count becomes worse, while in virtually all realistic examples, speed improvements are obtained.

To obtain the gate count of an unoptimized network, the gate counts of the macromodules used in that network were added up. The gate count of the optimized network was obtained from the AND/OR realization that Yun's tool [21] produces. In this table, the circuits *Call-C Idiom*, *Simple GVI* (parts 1 and 2), *Control-Block Sharing*, and *Call3-Merge* are various networks produced by the Occam or SHILPA compilers, and *decision wait* is a primitive similar to a generalized $C$-element. Notice that our optimizer achieves significant optimization for completion trees.

Another experiment we ran focused on the size of the peephole. As shown in Fig. 5, a modulo-17 counter was realized using a modulo-3 counter as a primitive. We ran two experiments: in the first, the modulo-3 counters were peephole optimized, resulting in an effective gate-count reduction from 23 to 22 gates. Next, the modulo-17 counter was peephole optimized. This resulted in an increase of gate-count from 66 to 309 gates. In other words, the burst-mode machine realization of the modulo-17 behavior proved to be quite inefficient in terms of gate count. This is an extreme example of what one can expect during peephole optimization and is also related to the details of operation of the version of the burst-mode machine synthesis algorithm being used. Although, in practice, such increases in size due to "optimization" are expected to be rare, it is still necessary to pick the correct grain size during peephole optimization. In our ongoing work, burst-mode reduction is employed during high-level synthesis to map control graphs into burst-mode controllers. In this context, grain-size control is achieved by means of a decomposition algorithm for Petri-net representations of control-flow graphs [30].

The reported speed estimates in Fig. 3 are *cycle time* [31] values, obtained by supplying an environment for the circuit that generates the next set of inputs to the circuit as soon as the circuit produces outputs corresponding to the current set of inputs. Speed measurements were done using a unit-delay simulator; more realistic examples, as well as measurement techniques, are under exploration. In practice, one may carry out macromodule subnetwork replacement until the required degree of performance is achieved. At this point, one may leave some macromodules (at the "top level") unoptimized. This can contribute toward maintaining the overall control organization of the system at a more intuitive level.

## APPENDIX
## HIDING AND PARALLEL COMPOSITION PRESERVE DI: PROOF SKETCH

A delay-insensitive module is one whose behavior remains unchanged (falls under the $\preceq$ relation below) when *arbitrary delays* are attached to all its output terminals (Dill's **DI** operator [25, p. 74]). Dill's hiding operator can be viewed as a special case of the **DI** operator where *infinite* delays are attached to all hidden output symbols (thus, preventing the environment from knowing anything about the activity on these symbols) and removing these symbols from the output alphabet of the module. A proof sketch for parallel composition is as follows. Suppose two DI modules $a$ and $b$ are being composed in parallel. Since $a$ and $b$ are DI, we have $\mathbf{DI}(a) \preceq a$ and $\mathbf{DI}(b) \preceq b$ where $\preceq$ is Dill's conformance operator. Since $\preceq$ is monotonic with respect to $\|$, we have $\mathbf{DI}(a) \parallel \mathbf{DI}(b) \preceq a \parallel b$. We now need to show $\mathbf{DI}(a \parallel b) \preceq a \parallel b$. Due to the transitivity of $\preceq$, it suffices to show that $\mathbf{DI}(a \parallel b) \preceq \mathbf{DI}(a) \parallel \mathbf{DI}(b)$. This is the case because $\mathbf{DI}(a \parallel b)$ is a special case of $\mathbf{DI}(a) \parallel \mathbf{DI}(b)$, where the delays attached to the "internal wires" (going from the outputs of $a$ to the inputs of $b$ and vice versa) are set to zero.

## ACKNOWLEDGMENT

The authors would like to thank N. Michell and S. Nowick for valuable discussions, and K. Yun for help with the 3-D tool.

## REFERENCES

[1] Y. N. Patt, S. J. Patel, M. Evers, D. H. Friendly, and J. Stark, "One billion transistors, one uniprocessor, one chip," *IEEE Trans. Comput.*, vol. 30, pp. 51–58, Sept. 1997.
[2] G. A. Pratt and J. Nguyen, "Distributed synchronous clocking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, pp. 314–328, Mar. 1995.
[3] J. Montanaro *et al.*, "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor," *Digital Tech. J.*, vol. 9, no. 1, pp. 49–62, 1997.
[4] A. Davis, B. Coates, and K. Stevens, "The post office experience: Designing a large asynchronous chip," in *Proc. Hawaii Int. Conf. Syst. Sci.*, vol. I. Piscataway, NJ: IEEE Comput. Soc. Press,1993, pp. 409–418.
[5] A. Marshall, B. Coates, and P. Siegel, "Designing an asynchronous communications chip," *IEEE Design & Test of Computers*, vol. 11, no. 2, pp. 8–21, 1994.
[6] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, F. Schalij, and R. van de Wiel, "A single-rail re-implementation of a DCC error detector using a generic standard-cell library," in *Asynchronous*

*Design Methodologies*. Piscataway, NJ: IEEE Comput. Soc. Press, 1995, pp. 72–79.
[7] A. Davis and S. M. Nowick, "Asynchronous circuit design: Motivation, background, & methods," in *Asynchronous Digital Circuit Design*, C. J. van Rijsbergen, Ed.. Berlin, Germany: Springer-Verlag, 1995, pp. 1–49.
[8] J. Haans, K. van Berkel, A. Peeters, and F. Schalij, "Asynchronous multipliers as combinational handshake circuits," in *Proc. IFIP Working Conf. Asynchronous Design Methods*, Manchester, U.K., Mar. 31–Apr. 2, 1993.
[9] E. Brunvand and R. F. Sproull, "Translating concurrent programs into delay-insensitive circuits," in *Int. Conf. Comput. Design*, Nov. 1989, pp. 262–265.
[10] V. Akella and G. Gopalakrishnan, "SHILPA: A high-level synthesis system for self-timed circuits," in *Int. Conf. Computer-Aided Design, ICCAD'92*, Nov. 1992, pp. 587–591.
[11] S. M. Ornstein, M. J. Stucki, and W. A. Clark, "A functional description of macromodules," in *Spring Joint Computer Conf.*, AFIPS, 1967.
[12] I. Sutherland, "Micropipelines," *Commun. ACM*, June 1989.
[13] E. Brunvand, "A cell set for self-timed design using actel FP-GA's," Dept. Comput. Sci., Univ. Utah, Salt Lake City, UT, Tech. Rep. 91-013, 1991.
[14] E. Brunvand, "Using FPGA's to implement self-timed systems," *J. VLSI Signal Processing*, (*Special Issue on Field Programmable Logic,*) vol. 6, 1993.
[15] J. T. Udding, "A formal model for defining and classifying delay-insensitive circuits and systems," *Distributed Computing*, no. 1, pp. 197–204, 1986.
[16] E. Brunvand, "Translating concurrent communicating programs into asynchronous circuits," Ph.D. dissertation, Carnegie-Mellon Univ. Pittsburgh, PA, 1991.
[17] K. van Berkel, "Handshake circuits: An intermediary between communicating processes and VLSI," Ph.D. dissertation, Philips Res. Lab., Eindhoven, The Netherlands, 1992.
[18] A. Davis, B. Coates, and K. Stevens, "The post office experience: Designing a large asynchronous chip," T. N. Mudge, V. Milutinovic, and L. Hunter, Eds., in *Proc. 26th Annu. Hawaiian Int. Conf. Syst. Sci.*, vol. 1, Jan. 1993, pp. 409–418.
[19] S. Nowick, "Automatic synthesis of burst-mode asynchronous controllers," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, 1993.
[20] S. M. Nowick, K. Y. Yun, and D. L. Dill, "Practical asynchronous controller design," in *Proc. Int. Conf. Comput. Design*, Oct. 1992, pp. 341–345.
[21] K. Y. Yun, D. L. Dill, and S. M. Nowick, "Synthesis of 3d asynchronous state machines," in *Proc. Int. Conf. Comput. Design*, Oct. 1992, pp. 346–350.
[22] S. Nowick, M. Dean, D. Dill, and M. Horowitz, "The design of a high-performance cache controller: A case study in asynchronous synthesis," T. N. Mudge, V. Milutinovic, and L. Hunter, Eds., in *Proc. 26th Annu. Hawaiian Int. Conf. Syst. Sci.,* vol. 1, Jan. 1993, pp. 419–427.
[23] G. Gopalakrishnan, N. Michell, E. Brunvand, and S. M. Nowick, "A correctness criterion for asynchronous circuit verification and optimization," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1309–1318, Nov. 1994.
[24] A. J. Martin, "Programming in VLSI: From communicating processes to delay-insensitive circuits," in *UT Year of Programming Institute on Cocurrent Programming*, C. A. R. Hoare, Ed. Reading, MA: Addison-Wesley, 1989.
[25] D. L. Dill, *Trace Theory for Automatic Hierarchical Verification of. Speed-Independent Circuits* (An ACM Distinguished Dissertation). Cambridge, MA: MIT Press, 1989.
[26] S. H. Unger, *Asynchronous Sequential Switching Circuits*. New York: Wiley, 1969.
[27] R. F. Sproull and I. E. Sutherland, *Asynchronous Systems, Vol. I: Introduction*. Palo Alto, CA: Sutherland, Sproull and Associates, 1986
[28] D. L. Dill, S. M. Nowick, and R. F. Sproull, "Specification and automatic verification of self-timed queues," *Formal Methods in Syst. Design*, vol. 1, no. 1, pp. 29–62, July 1992.
[29] E. Sternheim, R. Singh, and Y. Trivedi, *Digital Design with Verilog HDL*. Cupertino, CA: Automata 1990.
[30] P. Kudva, G. Gopalakrishnan, and H. Jacobson, "A technique for synthesizing distributed burst-mode circuits," in *Proc. 33th ACM/IEEE Design Automation Conf.*, Las Vegas, NV, June 1996, pp. 67–70.
[31] S. Burns and A. Martin, "Performance analysis and optimization of asynchronous circuits," in *Advanced Res. VLSI: Proc. 1991 Univ. California Santa Cruz Conf.*, C. Sequin, Ed. Cambridge, MA: MIT Press, 1991, pp. 71–86.

**Ganesh Gopalakrishnan** (S'82–M'85) received the Ph.D. degree in computer science from the State University of New York at Stony Brook, in 1986.

He is currently an Associate Professor in the Department of Computer Science, University of Utah, Salt Lake City. His research interests include asynchronous and self-timed circuit and system design, and formal hardware verification and design methods.

**Prabhakar Kudva** (S'91–A'96) received the Ph.D. degree in computer science from the University of Utah, Salt Lake City.

Since January 1995, he has been with the logic synthesis group at the IBM Thomas J. Watson Research Center, where he currently works on synthesis methodologies for advanced technologies, as well as placement driven synthesis. His research interests include asynchronous design.

Dr. Kudva is a member of the program committee for the 1999 IEEE Symposium on Asynchronous Circuits and Systems. He received an IBM Research Division Award in 1998 for his contributions to the design and realization of the Alliance microprocessor.

**Erik Brunvand** received the Ph.D. in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1991.

He is currently an Associate Professor in the Department of Computer Science, University of Utah, Salt Lake City. His research interests include asynchronous and self-timed circuit and system design, and automatic compilation of these systems from program descriptions. He is also interested in computer architecture, very large-scale integration (VLSI), and the interaction between them.